SUBMITTED TO: FTCS22, BOSTON, MASSACHUSETTS, JULY 8-10, 1992

COMPILER-ASSISTED STATIC CHECKPOINT INSERTION [1]

Junsheng Long and W. Kent Fuchs

Center for
Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois
Urbana, IL 61801

Jacob A. Abraham

Computer Engineering Research Center
Department of
Electrical and Computer Engineering
University of Texas at Austin
Austin, TX 78712

fuchs@crhc.uiuc.edu
(217) 333-9731
FAX: (217) 244-5686

Principal contact: W. Kent Fuchs

ABSTRACT

This paper describes a compiler-assisted approach for static checkpoint insertion. Instead of fixing the checkpoint location before program execution, a compiler enhanced polling mechanism is utilized to maintain both the desired checkpoint intervals and reproducible checkpoint locations. The technique has been implemented in a GNU CC compiler for Sun 3 and Sun 4 (Sparc) processors. Experiments demonstrate that the approach provides for stable checkpoint intervals and reproducible checkpoint placements with performance overhead comparable to a previously presented compiler-assisted dynamic scheme (CATCH) utilizing the system clock [17].

Key Words: *static checkpoints, checkpoint placement, checkpoint interval and compilers.*
Word Count: 3893.
Approval for submission and presentation has been obtained.

[1]This research is supported in part by the Department of the Navy and managed by the Office of the Chief of Naval Research under Contract N00014-91-J-1283, and in part by the National Aeronautics and Space Administration (NASA) under Contract NAG 1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS).

I. INTRODUCTION

Checkpointing and rollback is a common recovery strategy in fault-tolerant systems [1]. Considerable theoretical research has been devoted to determining optimal checkpoint intervals [2-7]. A practical problem in implementing checkpointing and rollback recovery is the maintenance of the desirable checkpoint interval. Checkpoints may be static in the sense that they are at fixed locations in a program or they may be dynamic such that their locations in a program may vary, as a function of time or system behavior. Although dynamic checkpoints can be implemented with existing hardware interrupt support (system clock), they are not reproducible. Static checkpoints must rely on either insertion of checkpoints before program execution or monitoring the program behavior during execution. Reproducible checkpoint intervals, as obtained with static checkpoints, can be used for debugging [8-11] or error detection by means of checkpoint comparison with replicated processes [12-13].

Chandy and Ramamoorthy have developed a scheme for application level checkpoint insertion, given a computation sequence, execution time, checkpoint time and restart time [14]. Their scheme is a graph-theoretic method to determine the optimal locations for checkpoint placement. Toueg and Balaoglu, and Upadhyaya and Saluja followed a similar approach [3, 15-16]. Li and Fuchs have studied techniques for checkpoint placement at the compiler level (*CATCH*) [17]. Checkpoint subroutines are transparently inserted in the user program by the compiler. *CATCH* is a dynamic checkpoint insertion scheme. To maintain the desirable checkpoint interval, the real time clock is polled to decide if a checkpointing call is due. Polling the real time clock can result in different checkpoint locations for different execution runs of the same computation due to the clock granularity (one second in Unix) and the workload on the system.

This paper presents a compiler-assisted approach for *static* checkpoint insertion. Instead of

fixing the checkpoint locations before program execution, a compiler enhanced polling mechanism is utilized to maintain both the desired checkpoint intervals and reproducible checkpoint locations. Instruction-based time measures are used to track the computation progress and thus checkpoint intervals. These measures produce static checkpoints by eliminating the real time clock. This approach has been implemented in a GNU CC compiler for Sun 3 and Sun 4 (SPARC) processors [18]. Experiments demonstrate that our approach provides for scalable checkpoint intervals and reproducible checkpoint placements with a performance overhead that is less than that of the previously presented compiler-assisted dynamic scheme (*CATCH*).

Section II describes our static checkpoint insertion approach and implementation. Section III discusses the experimental results.

II. Static Checkpoint Insertion

A. Instruction-based Time Measure

Maintaining desirable checkpoint intervals requires a time measure. Using the elapsed time of a computation as the time measure leads to dynamic checkpoints. This is because the elapsed time for a computation often varies from execution to execution due to resource sharing with other computations. Static checkpoint insertion requires a time measure that is independent of the real time clock and that describes checkpoint interval in terms of computation progress. Instruction-based measures, such as the instruction cycle count, satisfy both requirements, as they are only related to the instructions executed in a computation.

In this paper, we consider three architecture-independent instruction-based measures: instruction count, loop/function count and selected loop/function count. The instruction count (IC) is

the number of instructions in a computation, while the loop/function count (LFC) is the number of loop iterations and function calls. The selected loop/function count (SLFC) is the number of loop iterations/function calls for selected loops and functions. Although LFC and SLFC are potentially less accurate than ICC and IC with respect to the computation time, they can be maintained with low cost. The accuracy may still be adequate if the checkpoint interval contains a lot of loop iterations so that a stable mix of instructions is executed in each checkpoint interval.

B. Checkpoint Insertion Schemes

We use a polling mechanism with instruction-based time measures to accomplish the static checkpoint insertion. The compiler calculates the instruction-based time along an execution path. These statically calculated values for the time measure are accumulated in a counter during the program execution on the fly. The accumulated counter gives the time measure since the last checkpoint. The base compiler that was selected to implement our static checkpoint insertion is the GNU CC compiler version 1.40 for Sun 3 and Sun SPARC. A register transfer language (RTL) filter is placed between parsing and object code generation.

Based on the location of the time measure accumulation and polling points, the four schemes we have implemented are described below:

1. **B-B**: This scheme measures the instruction count (IC). The code for both the time measure accumulation and polling is inserted in each basic block of the program. A basic block is a sequence of consecutive instructions in which the program control enters at the top and leaves from the bottom with no branches or halts inside. Basic blocks in this paper are described in terms of RTL instructions.

2. **B-L**: In this scheme, the time measure is also the instruction count (IC). The time measure accumulation code is inserted in each basic block, while that for polling is placed in each loop.

3. **L-L**: This scheme uses the loop/function count (LFC) as the time measure. The code for the time measure accumulation and polling is inserted in every loop and function.

4. **SL-SL**: In this scheme, the time measure is the selected loop/function count (SLFC). The code for the time measure accumulation and polling is inserted only in the selected loops/functions.

C. SLFC Determination

In order to implement the SL-SL scheme, a method for selecting loops for SLFC was developed. Our approach is profile-based. Probe routines are placed into a program by the compiler. These probes collect the trace information during program profiling. The information collected is used to aid the loop selection for the SLFC measure. Once SLFC is determined, the compiler places static checkpoints in the program according to the SLFC measure.

There are two problems involved in selecting an SLFC measure: (1) to identify a set of loops that tend to appear throughout the execution trace, and (2) to determine a threshold value for each selected loop. This threshold value is important as the on-the-fly accumulated SLFC value is compared against this threshold value at each polling point in order to make a checkpoint decision. During profiling execution, each probe records the loop/function ID and calculates the frequency of occurrences of this loop in a checkpoint interval. If a set of loops can be found such that every checkpoint interval contains at least one loop from the loop set, this loop set may be a candidate for SLFC. The frequency associated with each loop for a checkpoint interval can be used as the threshold value for the loop.

Given a program and its profile data, the SLFC selection can be formulated as a cover set problem in a weighted bipartite graph. The checkpoint intervals and loop/function IDs are two sets of vertices. If a loop appears in a checkpoint interval, there is an edge between the checkpoint interval vertex and the loop vertex. The frequency of the loop occurrences in the checkpoint interval is the weight for this edge. The cover range of a loop vertex is the set of all the checkpoint interval vertices that are connected to the loop vertex. An SLFC cover set is a set of the loops such that their cover range contains all the checkpoint interval vertices.

There are four criteria for selecting a good SLFC cover set that gives a stable checkpoint interval with a small polling overhead:

- *Minimal overlapping*: The overlapping of cover ranges for two selected loops may result in unstable checkpoint intervals due to the interference of their possibly different threshold values.

- *Minimal cover set*: The size of an SLFC cover set is directly related to the code size overhead as the code inserted is proportional to the size of the cover set. Given that code size is not a problem for most applications, this criterion may be discounted during the selection of an SLFC cover set.

- *Minimal average frequency*: The average frequency for a loop in the SLFC cover set is used as the threshold value, for this loop, in our current implementation. A higher frequency leads to more frequent execution of the inserted checkpoint polling code for this loop and thus a higher run-time overhead.

- *Uniform Frequency*: This calls for a small variance in the frequencies for a loop in the SLFC cover set. As checkpointing is delayed for small frequency edges and is too frequent for large

frequency edges, large variance in frequency weights results in a more unstable checkpoint interval.

Although finding a minimal cover set is NP-complete, finding a cover set with minimal and uniform frequency can be mapped into the problem of finding a minimal total weight cover set. In the current implementation, a heuristic algorithm is used to combine all these criteria for SLFC selection (Figure 1). This heuristic is a greedy algorithm with different priorities for cover range, frequency average, and frequency variance. It selects loop vertices with large cover ranges and small frequencies under constraints of small relative frequency variance and little overlap for the selected loops.

III. Experimental Evaluation

Six benchmark programs were used to examining our static insertion technique. Our objective was to study effectiveness of the checkpoint interval maintenance in terms of:

1. The average checkpoint interval and its variance. This gives the effectiveness of an instruction-based time measure for checkpoint interval maintenance. A small variance implies that the instruction-based measure is accurate with respect to execution time.

2. Scalability of the checkpoint interval with respect to the instruction-based time measure threshold, for checkpoint polling tests. Linearity in the checkpoint interval with respect to the polling threshold allows for accurate prediction of the desired threshold.

3. The overhead for checkpoint interval maintenance due to the compiler-assisted technique. This overhead results from the time measure accumulation and checkpoint decision making at polling points.

```
select
   [1] the number of checkpoint intervals that a loop
       covers as the primary key (in decreasing order);
   [2] the average frequency of a loop as the secondary
       key (in increasing order); and
   [3] the relative standard deviation in frequency
       for a loop (std. dev./average) as the third
       key (in increasing order).
sort the vertices according to the above keys.

cover_set = NULL;

/* set for no overlapping cover range */
overlapping_size = 0;

while (size(cover_set) < desired_coverage) do
{
     for each vertex v in the sorted loop_set do
     {
          /* select a v with uniform frequency */
          if (freq_variance(v) > threshold) continue;

          if (size(cover_range(v) and cover_set) <=  overlapping_size)
               add v to cover_set;

          if (size(cover_set) >= desired_coverage) break;
     }
     /* relax the overlapping constraint */
     overlapping_size++;

     if (no changes in cover_set) break;
}
```

Figure 1. Heuristic SLFC Selection Algorithm.

4. Code size. This reflects the space overhead due to code insertions.

A. Benchmark Programs

Of the six benchmark programs we examined, four are scientific applications where loops are large and the calling depth is small. The other two programs contain a number of small loops and a large calling depth. The six benchmark programs are as follows:

convlv: is an FFT algorithm that finds the convolution of 1024 signals with one response [13, 17].

espresso: is a SPEC integer program for boolean function minimization, developed at the University of California at Berkeley [19]. It contains a lot of short loops, and recursive functions.

li: is a Lisp interpreter solving the 8-queen problem. It is a SPEC integer program developed by Sun Microsystems [19].

ludcmp: is an LU decomposition algorithm that decomposes 100 randomly generated matrices of size that is uniformly distributed between 50 and 60 [13].

rkf: uses the Runge-Kutta-Fehlberg method for solving the ordinary differential equation $y' = x + y$, $y(0) = 2$. This is a floating-point intensive program with large loop bodies [13, 17].

rsimp: is the revised Simplex method, for solving the linear optimization problem for the *BRANDY* set, from the Argonne National Laboratory [13, 17].

Table 1 describes the structure of the six programs in terms of the basic blocks. The block size is the number of the RTL instructions in a basic block. The static program information is collected from the program, during compilation, while the dynamic information is collected from profiling during execution. The fact that **convlv** and **rkf** have large loop bodies is reflected in their large dynamic basic block sizes. Similarly **espresso** and **li** have small loop bodies (and thus small dynamic basic blocks). The basic block size has an important impact on the performance overhead required for checkpoint interval maintenance. Smaller basic blocks result in a higher checkpoint

Table 1. Benchmark Characteristics.

Program	Static Basic Block		Dynamic Basic Block	
	Total number	Avg. size (ins./block)	Total number (10^6)	Avg. size (ins./block)
convlv	128	5.89	13.49	9.87
espresso	9018	3.10	108.56	2.85
li	3077	2.43	149.27	2.32
ludcmp	96	3.52	20.87	4.95
rkf	33	4.72	4.289	7.64
rsimp	185	3.08	73.02	4.57

maintenance cost in B-B and B-L as the ratio of the inserted code to the basic block size is high.

B. Checkpoint Intervals

Table 2 summarizes the checkpoint intervals generated on a Sun 3/50 diskless workstation. The threshold value, **L**, is the number of RTL instructions that are executed before the next checkpoint for B-B and B-L, and the number of loop iterations for L-L and SL-SL.

For all six programs, the checkpoint interval generated is linearly scalable. **L** is program specific due to different block structures in different programs. For the same **L**, the floating point programs (e.g., **rkf.**) generate longer checkpoint intervals than the integer benchmarks (**espresso** and **li**). The linear scalability of the checkpoint interval makes it possible to produce a consistent checkpoint interval across different programs. For example, the first few polling points can compare the targeted checkpoint interval with those generated under the initial **L**. If they disagree, **L** can be adjusted according to this linearly scalable relationship to obtain the desired checkpoint interval.

The standard deviation in the checkpoint interval reflects the accuracy of the interval as maintained by the instruction-based measure. Table 2 compares the standard deviations of all the four schemes. Generally, the standard deviations are less than one third of their corresponding checkpoint interval averages. Statistically, the actual checkpoint interval would most likely be

Table 2. Checkpoint Interval Maintenance (Sun 3).

Program	Scheme	L	Interval Average (secs.)			Standard Deviation (secs.)		
			L	5L	10L	L	5L	10L
convlv	B-B	500,000	1.54	7.69	15.41	0.0209	0.0355	0.0528
	B-L	500,000	1.44	7.22	14.71	0.0347	0.0768	0.1040
	L-L	50,000	7.78	23.80	47.72	0.0654	0.1168	0.1535
	SL-SL	50,000	4.76	23.67	47.65	0.0960	0.2287	0.3427
espresso	B-B	500,000	0.83	4.17	8.34	0.0043	0.0474	0.1492
	B-L	500,000	0.79	3.97	7.93	0.0218	0.2284	0.6066
	L-L	500,000	4.87	24.04	48.07	1.6159	6.0640	10.7311
	SL-SL	500,000	3.89	17.81	37.27	3.5543	8.5735	13.9470
li	B-B	500,000	1.18	5.88	11.77	0.0002	0.0007	0.0007
	B-L	500,000	0.94	4.62	9.52	0.0031	0.0005	0.5168
	L-L	500,000	7.30	37.02	72.49	0.0187	0.3404	0.0344
	SL-SL	500,000	6.08	29.13	58.27	0.3896	0.1070	0.1564
ludcmp	B-B	500,000	1.44	7.19	14.38	0.1734	0.1552	0.1976
	B-L	500,000	1.34	6.73	13.45	0.1642	0.1556	0.2017
	L-L	50,000	2.00	10.06	20.11	0.1146	0.1287	0.1969
	SL-SL	50,000	1.89	9.49	18.71	0.1825	0.4040	0.3159
rkf	B-B	500,000	5.05	25.26	50.71	0.5713	2.4046	4.3954
	B-L	500,000	4.95	24.74	49.65	0.5684	2.3988	4.3822
	L-L	50,000	8.18	40.68	81.38	1.0644	3.9855	6.9957
	SL-SL	50,000	8.08	39.55	76.84	0.6678	1.4292	4.6366
rsimp	B-B	500,000	1.26	6.29	12.58	0.0228	0.0820	0.1601
	B-L	500,000	1.18	5.88	12.09	0.0204	0.0651	0.2833
	L-L	500,000	15.34	76.70	154.40	0.2102	1.0541	1.8698
	SL-SL	500,000	15.22	75.99	151.81	0.3683	0.7133	0.2399

Table 3. Interrupt Driven Dynamic Scheme (Sun 3).

Program	Threshold value (secs.)	Average number of checkpoints	Average interval (secs.)	Standard deviation (secs.)	Exec. time overhead (%)
convlv	5	64.6	4.93	0.0693	0.17
espresso	5	41.2	4.89	0.0890	0.03
li	5	672.5	4.99	0.0220	0.24
ludcmp	5	51.0	4.87	0.1072	0.21
rkf	5	81.0	4.98	0.0480	0.07
rsimp	5	146.2	4.99	0.0557	0.08

Table 4. Checkpoint Interval Maintenance (Sun 4).

Program	Scheme	L	Interval Average (secs.)			Standard Deviation (secs.)		
			L	5L	10L	L	5L	10L
convlv	L-L	50,000	0.43	2.13	4.23	0.0139	0.0412	0.0709
	SL-SL	50,000	0.42	2.09	4.18	0.0119	0.0258	0.0363
espresso	L-L	500,000	1.06	5.27	10.53	0.2490	0.9836	1.8720
	SL-SL	500,000	0.87	3.96	8.30	0.7575	1.8839	2.9514
li	L-L	500,000	1.94	9.70	19.41	0.0074	0.0090	0.0297
	SL-SL	500,000	1.65	8.26	16.53	0.0182	0.0319	0.0463
ludcmp	L-L	50,000	0.25	1.26	2.52	0.0256	0.0289	0.3707
	SL-SL	50,000	0.23	1.15	2.30	0.0290	0.0515	0.0449
rkf	L-L	50,000	1.09	5.47	10.94	0.2189	0.8943	1.6498
	SL-SL	50,000	1.08	5.38	10.78	0.2323	0.9276	1.6498
rsimp	L-L	500,000	1.90	9.50	19.00	0.0227	0.0575	0.0918
	SL-SL	500,000	1.78	8.88	17.80	0.0628	0.1975	0.3587

within two or three standard deviations of the average interval. As mentioned previously, small changes in checkpoint frequency from the optimal frequency have little effect on the performance of the optimal solution [2-7]. Using the loop iteration count in L-L and SL-SL does not noticeably decrease the checkpoint interval accuracy. This may result from the large threshold L value, since the large number of loop iterations between checkpoints likely leads to a stable mixture of instructions for each checkpoint interval. As a comparison, Table 3 shows a program-independent checkpoint interval as maintained by the dynamic interrupt scheme using the system real time clock.

The results for L-L and SL-SL on a Sun 4 SPARC IPC are given in Table 4. The checkpoint

interval for the programs with a lot of floating-point operations and large loop bodies (**rkf** and **convlv**) is significantly larger than for those with smaller loop bodies. The integer programs, especially **espresso** and **li**, generated comparable intervals. This suggests that **L** is less program specific for integer programs in a RISC machine than in a CISC machine, as the frequency of almost one instruction-per-cycle improves the accuracy of instruction count or loop count as a measure of execution time. However, the SUN SPARC checkpoint intervals for the integer benchmarks (**espresso** and **li**) are in the same order of magnitude as the floating programs with comparable loop sizes, while the SUN 3 checkpoint intervals for the same integer programs are one order of magnitude smaller. The increased checkpoint intervals for **espresso** and **li** on SUN SPARC can be explained by the lack of support for integer multiplication and division on SUN SPARC [20]. In fact, integer multiplication and division are implemented through software traps, and integer multiplication and division are frequently used for address manipulations in the integer benchmarks we examined. The discrepancies in checkpoint interval between programs with intensive floating point operations and those with intensive integer operations still exist for SUN SPARC, since the IPC SPARC implementation supports the floating point through an off-chip floating point unit.

C. Checkpoint Interval Maintenance Overhead

In Table 5 the execution overhead in B-B and B-L is generally around 20% for programs with moderate basic block size (**convlv**, **ludcmp**, **rkf** and **rsimp**) and more than doubles the execution time for programs with small basic block size (< 3 for **espresso** and **li**). This is expected since the instruction-based measure is updated in each basic block. A smaller basic block results in larger updating code with respect to the block size, and thus larger insertion overhead. In B-B, the checkpoint polling point is also inserted in each basic block. B-B has roughly twice the overhead

Table 5. Checkpoint Interval Maintenance Overhead (Sun 3).

Program	Scheme	Execution time (secs.)		# of RTL insns.		Executable size (K bytes)		Text seg. size (K bytes)	
convlv	original	360.31		790		32		16	
	B-B	414.41	15.01%	1274	61.27%	40	25%	24	50%
	B-L	388.80	7.91%	959	21.39%	40	25%	24	50%
	L-L	367.45	1.98%	848	7.34%	40	25%	24	50%
	SL-SL	363.62	0.92%	811	2.66%	40	25%	24	50%
espresso	original	217.52		35621		176		152	
	B-B	517.70	138.00%	71611	101.04%	440	150%	408	168%
	B-L	418.56	92.42%	47005	31.96%	328	86%	296	95%
	L-L	312.52	43.67%	38708	8.67%	208	18%	176	16%
	SL-SL	218.70	0.54%	36340	2.02%	184	5%	160	5%
li	original	3330.18		10459		104		80	
	B-B	8151.98	144.79%	22860	118.57%	200	92%	168	110%
	B-L	6481.24	94.62%	14736	40.89%	160	54%	128	60%
	L-L	4429.34	33.01%	11763	12.47%	120	15%	88	10%
	SL-SL	3343.68	0.41%	10595	1.30%	104	0%	80	0%
ludcmp	original	245.17		414		24		8	
	B-B	317.43	29.47%	809	95.41%	32	33%	16	50%
	B-L	297.08	21.17%	560	35.27%	32	33%	16	50%
	L-L	261.73	6.75%	477	15.22%	24	0%	8	0%
	SL-SL	245.19	0.01%	437	5.56%	24	0%	8	0%
rkf	original	416.37		188		24		8	
	B-B	434.68	4.36%	331	76.06%	24	0%	8	0%
	B-L	430.60	3.42%	235	25.00%	24	0%	8	0%
	L-L	424.44	1.94%	202	7.45%	24	0%	8	0%
	SL-SL	417.77	0.34%	198	5.56%	24	0%	8	0%
rsimp	original	678.23		724		24		8	
	B-B	843.36	24.35%	1488	105.52%	32	33%	16	50%
	B-L	796.66	17.46%	1011	39.64%	32	33%	16	50%
	L-L	731.96	7.92%	852	17.68%	32	33%	16	50%
	SL-SL	678.51	0.04%	764	5.52%	24	0%	16	50%

as B-L. The large value for the polling threshold **L** and small block size imply that the polling at each basic block is unnecessary if a fine grain checkpoint interval is not targeted. If additional hardware is available, an interrupt driven mechanism can be used to eliminate the high overhead in B-B and B-L. In fact, a hardware instruction (cycle) count register can be added as part of the process context. It can be decremented whenever an instruction is executed. Once it reaches zero, an interrupt for checkpointing can obtain a static checkpoint without any polling overhead.

Table 6. Checkpoint Interval Maintenance Overhead (Sun 4).

Program	Scheme	Execution time (secs.)		# of RTL insns.		Executable size (K bytes)		Text seg. size (K bytes)	
convlv	original	28.55		1297		40		24	
	L-L	29.83	4.48%	1401	8.02%	40	0%	24	0%
	SL-SL	28.57	0.07%	1308	0.85%	40	0%	24	0%
espresso	original	44.74		46810		256		232	
	L-L	55.95	25.06%	51572	10.17%	304	19%	272	16%
	SL-SL	44.78	0.09%	46821	0.02%	272	6%	240	5%
li	original	939.23		13796		144		112	
	L-L	1087.21	15.76%	16137	16.97%	168	17%	128	14%
	SL-SL	943.95	0.50%	13807	0.08%	152	6%	112	0%
ludcmp	original	31.11		638		24		8	
	L-L	33.98	9.23%	742	16.30%	32	33%	16	50%
	SL-SL	31.27	0.51%	649	1.72%	32	33%	16	50%
rkf	original	54.47		312		24		8	
	L-L	55.62	2.11%	337	8.01%	24	0%	8	0%
	SL-SL	54.79	0.59%	323	3.53%	24	0%	8	0%
rsimp	original	83.86		1114		32		16	
	L-L	94.07	12.18%	1309	17.50%	32	0%	16	0%
	SL-SL	83.87	0.01%	1125	0.99%	32	0%	16	0%

The execution overhead for L-L is relatively small for programs with large loop sizes. However, L-L may still result in high polling overhead for programs with small loops (**espresso** and **li**). The profile-based SL-SL produces the smallest execution overhead of the four schemes, by polling only at the selected loops. In fact, the overhead is less than one percent of the execution time.

The increase in program size on a Sun 3 due to code insertion is presented in Table 5. The executable file size and text segment in the executable file are aligned at an 8K page boundary. The change in the executable and text segment may not reflect the checkpoint insertion if there is an unused internal fragment and the inserted code is smaller than the fragment. The number of RTL instructions in a program may be a better indicator for describing the code size overhead. The space overhead follows the general pattern in the execution time overhead. L-L typically has a code overhead of 20 percent on a Sun 3/50, while SL-SL has a mere 5 percent code size overhead.

Similar results for L-L and SL-SL on a Sun SPARC IPC are given in Table 6. The execution

Table 7. SL-SL Profiling Summary.

Program	Loop set	Cover set	Threshold set	Coverage (%)	Analysis time (secs.)	
					Sun 3	Sun 4
convlv	{0-14}	{14}	{15}	100	1.9	0.8
espresso	{0-783}	{621}	{910}	94.2	10.1	4.3
li	{0-388}	{156}	{13}	100	72.8	32.9
ludcmp	{0-14}	{4}	{39}	100	3.5	1.5
rkf	{0-2}	{1}	{7100}	100	0.4	0.1
rsimp	{0-29}	{20}	{10}	100	1.4	0.6

overhead is reduced (by almost a half) for integer benchmark programs (**espresso** and **li**) and increased for the floating point programs for L-L. The execution time overhead for SL-SL is again less than one percent of total execution time. The space overhead for L-L on a Sun SPARC IPC is slightly increased due to the relatively large RISC code size compared to the non-RISC code size. The space overhead for SL-SL is less than four percent of program size.

D. Profiling and SLFC Selection

In our profiling experiments, the minimal coverage that was selected for the SLFC selection algorithm was 90 percent. Table 7 indicates that our algorithm identifies only one loop/function polling point for each of the six programs we considered. Tables 2 and 5 have shown that this SLFC selection is effective in reducing overhead and producing stable checkpoint intervals.

The key to a successful profiling is to use a representative data set during profiling. There are four sets of data for **espresso**. We used the first set (**bca.in**) as the profile data. Table 8 compares the results for the program profiled on **bca.in** and run with three non-profiled data sets. The execution overhead for SL-SL is still less than one percent. Although the produced checkpoint intervals are less than the profiled intervals, they are within the same order of magnitude. This indicates that **bca.in** may not be the representative data set for the four data sets, and it highlights

Table 8. SL-SL Results for Non-profiled Data Sets.

Data set	Scheme	Sun 3		Sun 4	
		Interval 10L	Exec. time (sec.)	Interval 10L	Exec. time (secs.)
bca.in	orginal		217.52		44.74
	SL-SL	37.27	218.70	8.30	44.78
cps.in	orginal		269.14		57.68
	SL-SL	17.71	269.52	3.76	57.72
ti.in	orginal		323.94		69.90
	SL-SL	10.08	324.28	2.15	70.02
tial.in	orginal		554.62		113.88
	SL-SL	26.08	555.48	5.28	114.40

the need for representative profiling data in using the profile-based SLFC selection.

E. Comparison with CATCH

With respect to overhead, the L-L scheme is very close to the basic *CATCH* [17]. The L-L run-time overhead is essentially the same as that for maintaining the potential checkpoint leverage in *CATCH*. The extra overhead for *CATCH* is in polling the real time clock. The results for SL-SL are comparable to those for the trained *CATCH*, as both use the profile-based approach. In the trained *CATCH*, the cover set is selected based on coverage and checkpoint size with no regard to the threshold value determination and non-overlapping of cover ranges. Table 9 compares L-L and SL-SL with their corresponding *CATCH* schemes. The interrupt-driven dynamic scheme is also presented. Generally, the overhead for our static scheme (L-L and SL-SL) is less than that for the dynamic *CATCH*. The overhead for SL-SL is comparable to that for the interrupt-driven dynamic approach, without using extra hardware support.

Table 9. Execution Time Overhead: Static vs. Dynamic Insertion.

Program	Static Insertion		Dynamic Insertion		
	L-L	SL-SL	CATCH		Interrupt
			Basic	Trained	Driven
convlv	1.98%	0.92%	4.76%	1.39%	0.17%
espresso	43.67%	0.54%	56.52%	9.85%	0.03%
li	33.01%	0.41%	38.12%	6.11%	0.24%
ludcmp	6.75%	0.01%	8.18%	3.76%	0.21%
rkf	1.94%	0.34%	2.74%	0.75%	0.07%
rsimp	7.92%	0.04%	13.21%	5.22%	0.08%

IV. Summary

In this paper, a compiler-assisted approach for static checkpoint insertion has been presented. This approach uses an instruction-based measure to describe checkpoint intervals in terms of computation progress. The instruction-based measure is independent of the real time clock, although it has a time attribute related to the program execution. This relationship between computation progress and execution time makes it possible to use an instruction-based measure for checkpoint interval maintenance.

Four different schemes, based on this approach, have been implemented and evaluated. Experiments show that our static method can generate a stable and scalable checkpoint interval. The overhead for the basic block-based schemes, such as B-B and B-L, is high without hardware support. The loop iteration count based scheme L-L can obtain a checkpoint interval comparable to B-B and B-L, with significantly less overhead. The block size of a program has an important impact on insertion overhead for our schemes. The profile-based SL-SL scheme can effectively reduce both the run-time overhead as well as the space overhead. In fact, this scheme can produce scalable and stable checkpoint intervals with an overhead comparable to that of the hardware interrupt scheme. This only requires a representative data set for accurate prediction of program run time behavior.

REFERENCES

[1] P. A. Lee and T. Anderson, *Fault Tolerance: principles and practice.* Springer-Verlag/Wien, 1990.

[2] P. L'Ecuyer and J. Mallenfant, "Computing optimal checkpointing strategies for rollback and recovery systems," *IEEE Trans. on Computers*, vol. 37, pp. 491–496, April, 1988.

[3] S. Toueg and O. Babaoglu, "On the optimum checkpoint selection problem," *SIAM Journal on Computing*, vol. 13, pp. 630–649, Aug., 1984.

[4] C. M. Krishna, K. G. Shin, and Y.-H. Lee, "Optimization criteria for checkpoint placement," *CACM*, vol. 27, no. 6, pp. 1008–1012, Oct. 1984.

[5] A. Duda, "The effects of checkpointing on program execution time," *Information Processing Letters*, vol. 16, pp. 221–229, 1983.

[6] E. Gelenbe and D. Derochette, "Performance of rollback recovery systems under intermittent failures," *CACM*, vol. 21, no. 6, pp. 493–499, 1978.

[7] J. W. Young, "A first order approximation to the optimal checkpoint interval," *CACM*, vol. 17, no. 9, pp. 530–531, Sept. 1974.

[8] S. Feldman and C. Brown, "A system for program debugging via reversible execution," *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, vol. 24(1), pp. 112–123, Jan. 1989.

[9] D. Pan and M. Linton, "Supporting reverse execution for parallel programs," *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, vol. 24(1), pp. 124–129, Jan. 1989.

[10] L. D. Wittie, "Debugging distributed C programs by real time replay," *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, vol. 24(1), pp. 57–67, Jan. 1989.

[11] K. Li, J. F. Naughton, and J. S. Plank, "Real-time, concurrent checkpoint for parallel programs," *Proc. 2nd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pp. 79–88, March, 1990.

[12] J. Long, W. K. Fuchs, and J. A. Abraham, "A forward recovery strategy using checkpointing in parallel systems," *Proc. Int'l. Conf. on Parallel Processing*, vol. 1, pp. 272–275, 1990.

[13] J. Long, W. K. Fuchs, and J. A. Abraham, "Implementing forward recovery using checkpointing in distributed systems," *Proc. 2nd IFIP Working Conf. on Dependable Computing for Critical Applications*, pp. 20–27, Feb. 1991.

[14] K. M. Chandy and C. V. Ramamoorthy, "Rollback and recovery strategies for computer programs," *IEEE Trans. on Computers*, vol. 21(6), pp. 546–556, June 1972.

[15] J. S. Upadhyaya and K. K. Saluja, "A watchdog processor based general rollback technique with multiple retries," *IEEE Trans. on Software Engineering*, vol. 12(1), pp. 87–95, Jan. 1986.

[16] J. S. Upadhyaya and K. K. Saluja, "An experimental study to determine task size for rollback recovery systems," *IEEE Trans. on Computers*, vol. 37(7), pp. 872–877, July 1988.

[17] C. C. Li and W. K. Fuchs, "Catch: Compiler-assisted techniques for checkpointing," *Proc. 20th Int'l. Symp. on Fault-Tolerant Computing Systems*, pp. 74–81, 1990.

[18] R. M. Stallman, "Using and porting GNU CC," *Proc. 2nd Int'l Conf. on Computers and Applications*, 1990.

[19] SPEC, *Spec newsletter.* Fremont, CA: SPEC, Feb. 1989.

[20] ROSS Technology, Inc., *SPARC RISC user's guide.* 7748 Hwy. 290 West, Austin, Texas 78736: ROSS Technology, Inc., 1990.